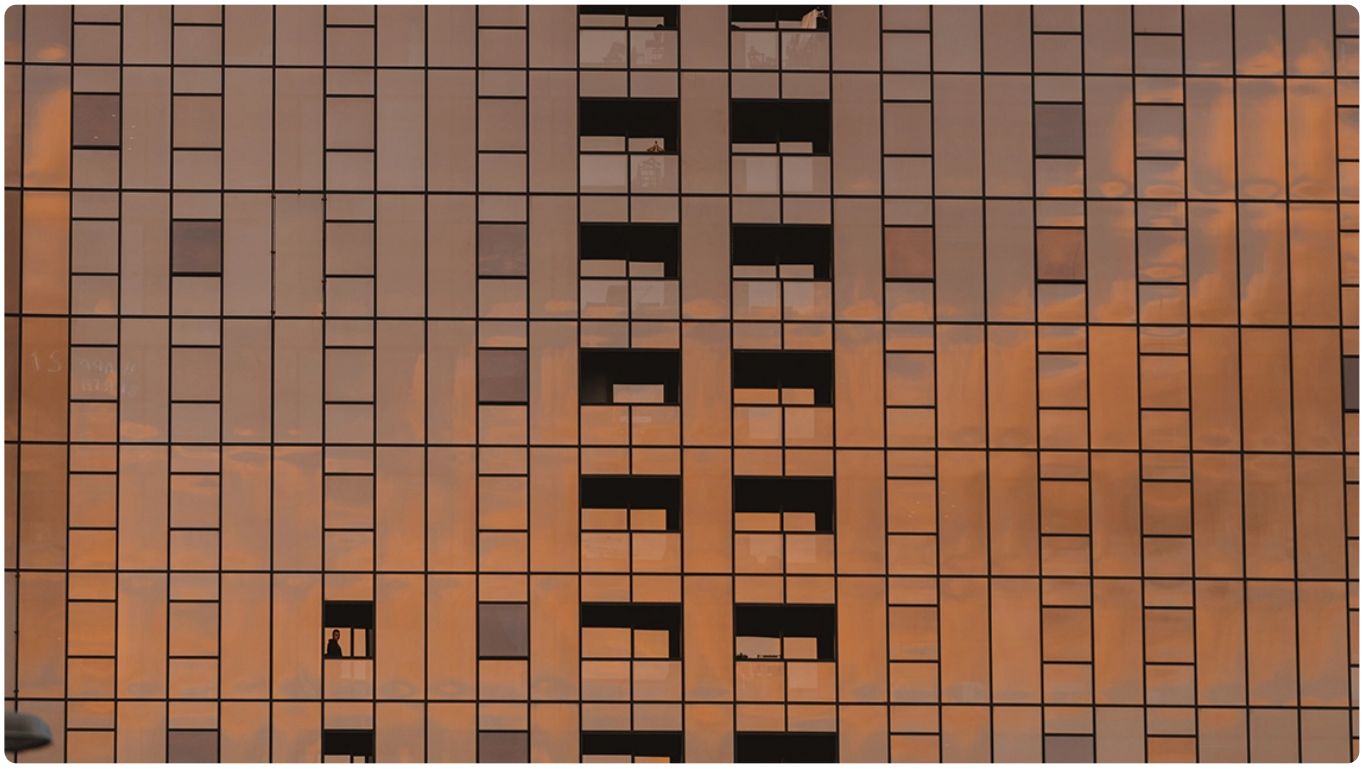
下载APP

01｜内存：值放堆上还是放栈上，这是一个问题

2021-08-16 陈天

《陈天 · Rust 编程第一课》 课程介绍 >



讲述：陈天
时长 15:20 大小 14.05M

你好，我是陈天。今天我们打卡 Rust 学习的第一讲。

你是不是已经迫不及待想要了解 Rust 了，但是别着急，我们不会按常规直接开始介绍语法，**而会先来回顾那些你平时认为非常基础的知识，比如说内存、函数。**

提到基础知识，你是不是已经有点提不起兴趣了，这些东西我都知道，何必浪费时间再学一次呢？其实不然，这些年我接触过的资深工程师里，基础知识没学透，工作多年了，还得回来补课的大有人在。

以最基础的内存为例，很多人其实并没有搞懂什么时候数据应该放在栈上，什么时候应该在堆上，直到工作中实际出现问题了，才意识到数据的存放方式居然会严重影响并发安全，无奈回头重新补基础，时间精力的耗费都很大。

其实作为开发者，我们一辈子会经历很多工具、框架和语言，但是这些东西无论怎么变，底层逻辑都是不变的。

所以今天我们得**回头重新思考，编程中那些耳熟能详却又似懂非懂的基础概念，搞清楚底层逻辑**。而且这些概念，对我们后面学习和理解 Rust 中的知识点非常重要，之后，我们也会根据需要再穿插深入讲解。

代码中最基本的概念是变量和值，而存放它们的地方是内存，所以我们就从内存开始。

内存

我们的程序无时无刻都在跟内存打交道。在下面这个把 “hello world!” 赋值给 s 的简单语句中，就跟只读数据段（RODATA）、堆、栈分别有深度交互：

```
1 let s = "hello world".to_string();
```

复制代码

你可以使用 Rust playground 里这个代码片段 感受一下字符串的内存使用情况。

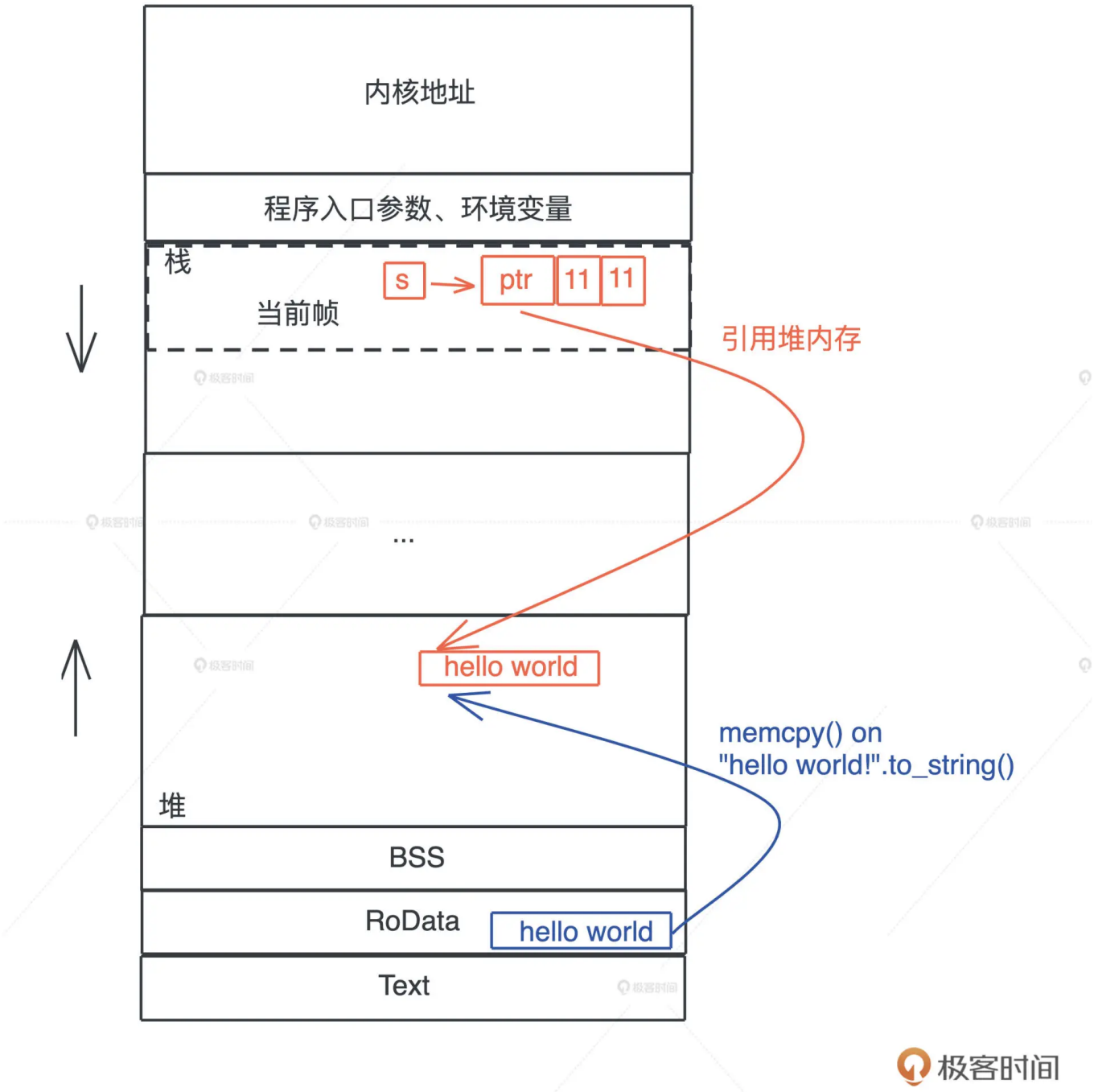
首先，“hello world” 作为一个字符串常量（string literal），在编译时被存入可执行文件的 .RODATA 段（GCC）或者 .RDATA 段（VC++），然后在程序加载时，获得一个固定的内存地址。

当执行 “hello world”.to_string() 时，在堆上，一块新的内存被分配出来，并把 “hello world” 逐个字节拷贝过去。

当我们把堆上的数据赋值给 s 时，s 作为分配在栈上的一个变量，它需要知道堆上内存的地址，另外由于堆上的数据大小不确定且可以增长，我们还需要知道它的长度以及它现在有多大。

最终，**为了表述这个字符串，我们使用了三个 word**：第一个表示指针、第二个表示字符串的当前长度（11）、第三个表示这片内存的总容量（11）。在 64 位系统下，三个 word 是 24 个字节。

你也可以看下图，更直观一些：



刚才提到字符串的内容在堆上，而指向字符串的指针等信息在栈上，现在就是检验你内存基础知识是否扎实的时候了：**数据什么时候可以放在栈上，什么时候需要放在堆上呢**？

这个问题，很多使用自动内存管理语言比如 Java/Python 的开发者，可能有一些模糊的印象或者规则：

基本类型（primitive type）存储在栈上，对象存储在堆上；

少量数据存储在栈上，大量的数据存储在堆上。

这些虽然对，但并没有抓到实质。如果你在工作中只背规则套公式，一遇到特殊情况就容易懵，但是如果明白公式背后的推导逻辑，即使忘了，也很快能通过简单思考找到答案，所以接下来我们深挖堆和栈的设计原理，看看它们到底是如何工作的。

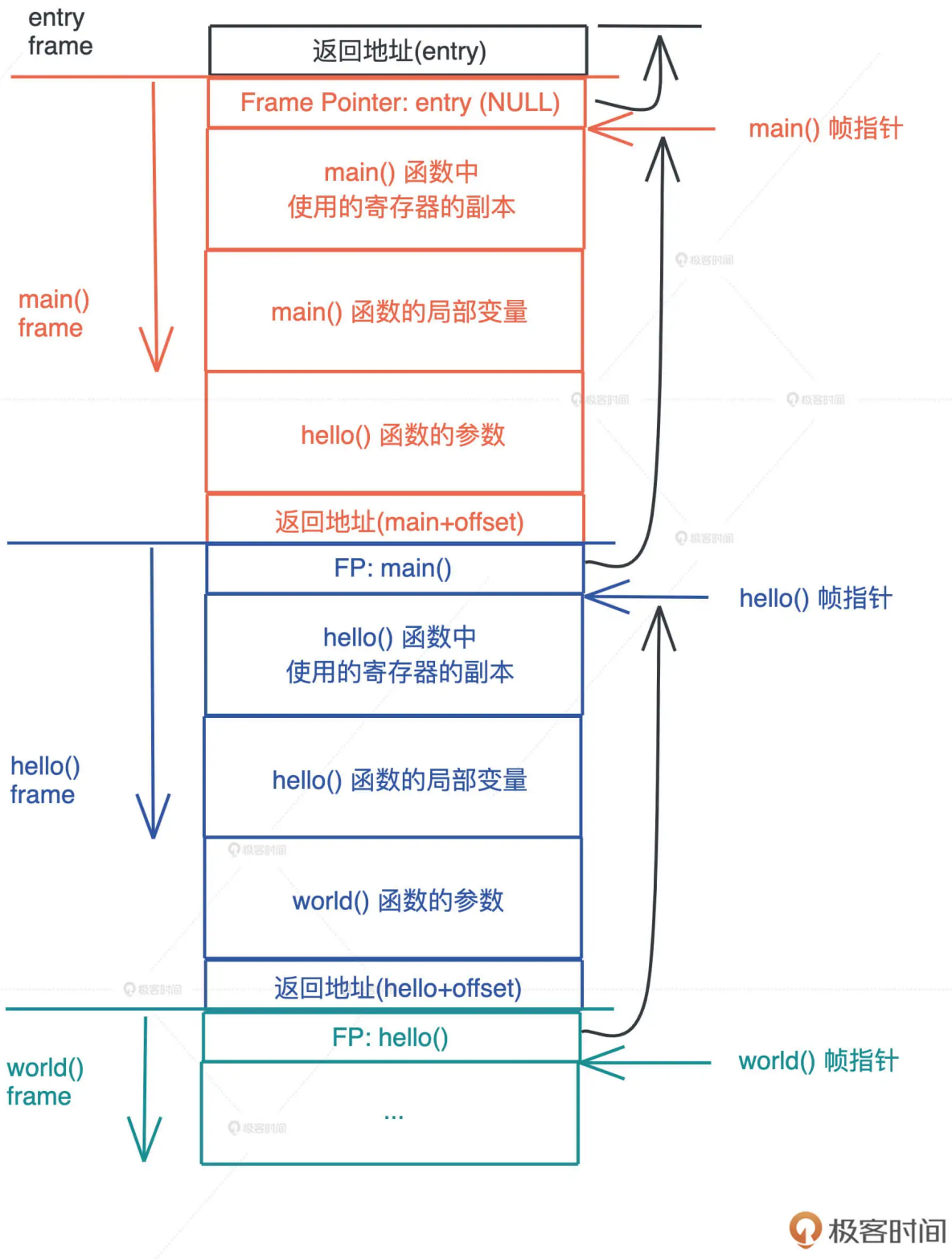
栈

栈是程序运行的基础。每当一个函数被调用时，一块连续的内存就会在栈顶被分配出来，这块内存被称为帧（frame）。

我们知道，栈是自顶向下增长的，一个程序的调用栈最底部，除去入口帧（entry frame），就是 main() 函数对应的帧，而随着 main() 函数一层层调用，栈会一层层扩展；调用结束，栈又会一层层回溯，把内存释放回去。

在调用的过程中，**一个新的帧会分配足够的空间存储寄存器的上下文**。在函数里使用到的通用寄存器会在栈保存一个副本，当这个函数调用结束，通过副本，可以恢复出原本的寄存器的上下文，就像什么都没有经历一样。此外，函数所需要使用到的局部变量，也都会在帧分配的时候被预留出来。

整个过程你可以再看看这张图辅助理解：



那一个函数运行时，**怎么确定究竟需要多大的帧呢**？

这要归功于编译器。在编译并优化代码的时候，一个函数就是一个最小的编译单元。

在这个函数里，编译器得知道要用到哪些寄存器、栈上要放哪些局部变量，而这些都要在编译时确定。所以编译器就需要明确每个局部变量的大小，以便于预留空间。

这下我们就明白了：**在编译时，一切无法确定大小或者大小可以改变的数据，都无法安全地放在栈上，最好放在堆上。**比如一个函数，参数是字符串：

复制代码

```
fn say_name(name: String) {}

// 调用
say_name("Lindsey".to_string());
say_name("Rosie".to_string());
```

字符串的数据结构，在编译时大小不确定，运行时执行到具体的代码才知道大小。比如上面的代码，“Lindsey” 和 “Rosie” 的长度不一样，say_name() 函数只有在运行的时候，才知道参数的具体的长度。

所以，我们无法把字符串本身放在栈上，只能先将其放在堆上，然后在栈上分配对应的指针，引用堆上的内存。

放栈上的问题

从刚才的图中你也可以直观看到，栈上的内存分配是非常高效的。只需要改动栈指针（stack pointer），就可以预留相应的空间；把栈指针改动回来，预留的空间又会被释放掉。预留和释放只是动动寄存器，不涉及额外计算、不涉及系统调用，因而效率很高。

所以理论上说，只要可能，我们应该把变量分配到栈上，这样可以达到更好的运行速度。

那为什么在实际工作中，我们又要避免把大量的数据分配在栈上呢？

这主要是考虑到调用栈的大小，避免栈溢出（stack overflow）。一旦当前程序的调用栈超出了系统允许的最大栈空间，无法创建新的帧，来运行下一个要执行的函数，就会发生栈溢出，这时程序会被系统终止，产生崩溃信息。

过大的栈内存分配是导致栈溢出的原因之一，更广为人知的原因是递归函数没有妥善终止。一个递归函数会不断调用自己，每次调用都会形成一个新的帧，如果递归函数无法终

止，最终就会导致栈溢出。

堆

栈虽然使用起来很高效，但它的局限也显而易见。**当我们需要动态大小的内存时，只能使用堆**，比如可变长度的数组、列表、哈希表、字典，它们都分配在堆上。

堆上分配内存时，一般都会预留一些空间，这是最佳实践。

比如你创建一个列表，并往里添加两个值：

复制代码

```
let mut arr = Vec::new();
arr.push(1);
arr.push(2);
```

这个列表实际预留的大小是 4，并不等于其长度 2。这是因为堆上内存分配会使用 libc 提供的 malloc() 函数，其内部会请求操作系统的系统调用，来分配内存。系统调用的代价是昂贵的，所以我们要避免频繁地 malloc()。

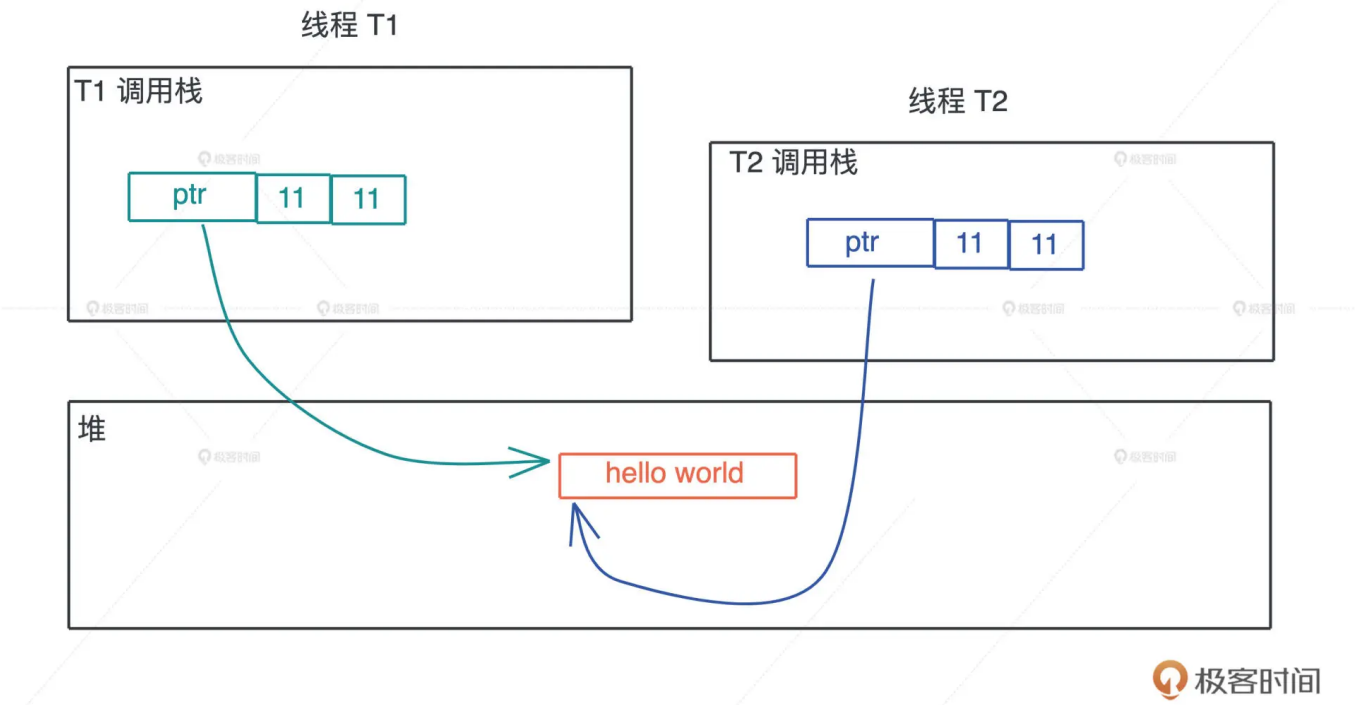
对上面的代码来说，如果我们需要多少就分配多少，那列表每次新增值，都要新分配一大块的内存，先拷贝已有数据，再把新的值添加进去，最后释放旧的内存，这样效率很低。所以在堆内存分配时，预留的空间大小 4 会大于需要的实际大小 2 。

除了动态大小的内存需要被分配到堆上外，**动态生命周期的内存也需要分配到堆上**。

上文中我们讲到，栈上的内存在函数调用结束之后，所使用的帧被回收，相关变量对应的内存也都被回收待用。所以栈上内存的生命周期是不受开发者控制的，并且局限在当前调用栈。

而堆上分配出来的每一块内存需要显式地释放，**这就使堆上内存有更加灵活的生命周期，可以在不同的调用栈之间共享数据**。

如下图所示：



放堆上的问题

然而，堆内存的这种灵活性也给内存管理带来很多挑战。

如果手工管理堆内存的话，堆上内存分配后忘记释放，就会造成**内存泄漏**。一旦有内存泄漏，程序运行得越久，就越吃内存，最终会因为占满内存而被操作系统终止运行。

如果堆上内存被多个线程的调用栈引用，该内存的改动要特别小心，需要加锁以独占访问，来避免潜在的问题。比如说，一个线程在遍历列表，而另一个线程在释放列表中的某一项，就可能访问野指针，导致**堆越界**（heap out of bounds）。而堆越界是第一大内存安全问题。

如果堆上内存被释放，但栈上指向堆上内存的相应指针没有被清空，就有可能**发生使用已释放内存（use after free）的情况**，程序轻则崩溃，重则隐含安全隐患。根据微软安全反应中心（MSRC）的研究，这是第二大内存安全问题。

GC、ARC 如何解决

为了避免堆内存手动管理造成的这些问题，以 Java 为首的一系列编程语言，采用了追踪式垃圾回收（Tracing GC）的方法，来自动管理堆内存。这种方式通过定期标记（mark）找出不再被引用的对象，然后将其清理（sweep）掉，来自动管理内存，减轻开发者的负担。

而 OjbC 和 Swift 则走了另一条路：自动引用计数（Automatic Reference Counting）。在编译时，它为每个函数插入 retain/release 语句来自动维护堆上对象的引用计数，当引用计数为零的时候，release 语句就释放对象。

我们来对比一下这两个方案。

从效率上来说，GC 在内存分配和释放上无需额外操作，而 ARC 添加了大量的额外代码处理引用计数，所以 GC 效率更高，吞吐量（throughput）更大。

但是，GC 释放内存的时机是不确定的，释放时引发的 STW（Stop The World），也会导致代码执行的延迟（latency）不确定。**所以一般携带 GC 的编程语言，不适于做嵌入式系统或者实时系统**。当然，Erlang VM是个例外， 它把 GC 的粒度下放到每个 process，最大程度解决了 STW 的问题。

我们使用 Android 手机偶尔感觉卡顿，而 iOS 手机却运行丝滑，大多是这个原因。而且做后端服务时，API 或者服务响应时间的 p99（99th percentile）也会受到 GC STW 的影响而表现不佳。

说句题外话，上面说的 GC 性能和我们常说的性能，涵义不太一样。常说的性能是吞吐量和延迟的总体感知，和实际性能是有差异的，GC 和 ARC 就是典型例子。GC 分配和释放内存的效率和吞吐量要比 ARC 高，**但因为偶尔的高延迟，导致被感知的性能比较差**，所以会给人一种 GC 不如 ARC 性能好的感觉。

小结

今天我们重新回顾基础概念，分析了栈和堆的特点。

对于**存入栈上的值，它的大小在编译期就需要确定**。栈上存储的变量生命周期在当前调用栈的作用域内，无法跨调用栈引用。

堆可以存入大小未知或者动态伸缩的数据类型。堆上存储的变量，其生命周期从分配后开始，一直到释放时才结束，因此堆上的变量允许在多个调用栈之间引用。但也导致堆变量的管理非常复杂，手工管理会引发很多内存安全性问题，而自动管理，无论是 GC 还是 ARC，都有性能损耗和其它问题。

一句话对比总结就是：**栈上存放的数据是静态的，静态大小，静态生命周期；堆上存放的数据是动态的，动态大小，动态生命周期**。

下一讲我们会讨论基础概念，比如值和类型、指针和引用、函数、方法和闭包、接口和虚表、并发与并行、同步和异步，以及 Promise/async/await ，这些我们学习 Rust 或者任何语言都会接触到。

思考题

最后，是课后练习题环节，欢迎在留言区分享你的思考。

1. 如果有一个数据结构需要在多个线程中访问，可以把它放在栈上吗？为什么？

2. 可以使用指针引用栈上的某个变量吗？如果可以，在什么情况下可以这么做？

另外，文中出现的所有参考资料链接，我都会再统一整理到文末的“拓展阅读”板块，所以非常推荐你先跟着文章的思路走，学完之后如果有兴趣，可以看看我分享给你的其他资料。

感谢你的收听，如果你觉得有收获，也欢迎你点“请朋友读”，分享给你身边的朋友，邀他一起讨论。我们下一讲见!

订阅后，戳这里加入“Rust 语言入门交流群”，一起来学习 Rust。

拓展阅读

1. 微软安全反应中心（MSRC）的研究

2. 追踪式垃圾回收 Tracing GC

3. 自动引用计数 Automatic Reference Counting

4. Erlang VM 把 GC 的粒度下放到每个 process，最大程度解决了 STW 的问题

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 32 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 开篇词｜让Rust成为你的下一门主力语言

精选留言 (27)

写留言

pedro 置顶

2021-08-16

1. 如果有一个数据结构需要在多个线程中访问，可以把它放在栈上吗？为什么？
不能，栈上的数据会随着当前线程的函数调用栈而回收，多个线程访问须在堆上开辟。

2. 可以使用指针引用栈上的某个变量吗？如果可以，在什么情况下可以这么做？
可以，在当前函数调用栈中，可以新建变量在栈上开辟，顺便分配一个指针指向它，但...
展开

作者回复: 嗯，@pedro 回答得很详尽了。我补充一下。1. 在多线程场景下，每个线程的生命周期是不固定的，无法在编译期知道谁先结束谁后结束，所以你不能把属于某个线程 A 调用栈上的内存共享给线程 B，因为 A 可能先于 B 结束。这时候，只能使用堆内存。这里有个例外，如果结束的顺序是确定的，那么可以共享，比如 scoped thread；2. 而同一个调用栈下，main() 调用 hello()，再调用 world()，编译器很清楚，world() 会先结束，之后是 hello()，最后是 main()。所以在 world() 下用指针引用 hello() 或者 main() 内部的变量没有问题，这个指针必然先于它指向的值结束。这个两个问题的实质是我们要搞明白哪些东西在编译期可以确定它们的关系或者因果，哪些只能在运行期确定。

6 37

有铭 置顶
2021-08-17

在编译时，一切无法确定大小或者大小可以改变的数据，都无法放在栈上，只能放在堆上
======
我对这句话感到困惑，那可变长参数呢？可变长参数也属于在编译时无法确认大小的，难道也放在堆上？我搜索了一下发现rust似乎不支持可变长参数的函数，也不支持函数重载。所以我在想这种堆栈的特性是否是rust独有的。因为至少，Java就是支持可变长参...
展开

作者回复: 嗯，这是个好问题。这句话我表述地太绝对了。应该修改为：「在编译时，一切无法确定大小或者大小可以改变的数据，都无法**安全地**放在栈上，**最好**放在堆上」。

可变参数的函数是一个很好的例子。对于 Java，可变参数 String... a 是 String[] 的语法糖，它是放在堆上的。在 C 语言里，这个行为是未定义的，它只是定义了你可以通过 var_start / var_end 来获得可变参数的起始位置，以及最终结束可变参数的访问。但 gcc 的实现将可变参数放在栈上（估计是为了性能）。比如 C，你可以用 var_start / var_end 获取可变参数，但如果不小心处理， 会导致访问栈上的垃圾内容，甚至导致程序崩溃：

```C
#include <stdio.h>
#include <stdarg.h>

int sum(int count, ...) {
    va_list ap;
    int i;
```

```
    double sum = 0;

    va_start(ap, count);
    for (i = 0; i < count; i++) {
        sum += va_arg(ap, int);
    }
    va_end(ap);

    return sum;
}

int main(int argc, char const *argv[]) {
    printf("%d\n", sum(10, 1, 2, 3)); // 传入 3 个值但 count 为 10
    return 0;
}
```

同时谢谢 c4f 的提醒，alloca() 可以在栈上分配动态大小的内存，然而使用它需要非常小心，按 linux 的文档（https://man7.org/linux/man-pages/man3/alloca.3.html），官方建议配合 longjmp 使用。alloca() 如果分配太大的数据，超过栈容量会导致程序崩溃，即使你分配很小的数据，但如果使用 alloca() 的函数被优化导致 inline，又恰巧出现在大的 for/while 循环中，也可能会导致崩溃。

所以，这两种在栈上分配可变大小的数据，是不安全的。

4 3

Christian 置顶

2021-08-17

1. 如果有一个数据结构需要在多个线程中访问，可以把它放在栈上吗？为什么？
这种情况是可能的。比如主线程创建一个子线程并等待子线程结束。子线程使用指针访问主线程当前栈帧上的数据。
2. 可以使用指针引用栈上的某个变量吗？如果可以，在什么情况下可以这么做？
可以，只要指针的生命周期小于等于引用的值就行。

展开

作者回复: 主线程创建子线程并等待子线程结束这个动作，是我们通过程序的上下文得出的结论，但这是运行时的编译器并不能理解他们的因果关系。比如：

```rust
fn main() {
```

```
    let x = 10;

    let handle = thread::spawn(|| {
        println!("x: {}", x);
    });
    handle.join().unwrap();
}
```

从代码中我们很容易得出结论，主线程的 x 被子线程引用是安全的，但在编译期编译器无法做出这样的保证。

1

神山 | 雷神山

2021-08-16

1. 如果有一个数据结构需要在多个线程中访问，可以把它放在栈上吗？为什么？
不能, 每个线程都会拥有自己的栈,栈上数据无法进行跨栈访问.
2. 可以使用指针引用栈上的某个变量吗？如果可以，在什么情况下可以这么做？
可以,只要指针的生命周期小于或者等于栈上的引用源就行,如果生命周期大于引用源就会出现野指针的情况.在rust中会报错从而无法编译.

展开

作者回复: 👍

3

chapin

2021-08-19

往往评论才是精华。能看到很多大神的回复。

展开

2

只吃无籽西瓜

2021-08-18

1.每个线程有独立的栈和寄存器，多线程共享的数据不能放在栈上。
2.可以使用指针，但是别把这个指针返回。

展开

1

拉斯特 酥

2021-08-17

打卡打卡，与编译器作斗争。(ง •̀_•́)ง

展开

1

halweg

2021-08-16

看了一讲，都不用看目录了，
这就是我想要的东西

作者回复: 多谢支持

1

特修斯之船

2021-08-19

对于大小未知的变量都放在堆中，那是不是可以认为动态语言例如JS和Python，变量都是放在堆中？

f

2021-08-19

看到老师文章中的playground的代码，想起来一直存在的一个疑惑，"{:p}"输出某个变量的地址，是这个变量自身结构的地址，还是变量所指向值的地址呢。比如
```rust
let s = "helloworld".to_string();
println!("addr: {:p}", &s);...
```
展开

1

旻言 酥

2021-08-19

我来说说我的想法：
1. 我认为可以，如果你的业务是在当前函数栈中，把栈上变量传给给worker线程池，或者自己启动多个worker线程处理，那么只要发起调用的线程同步等待worker线程处理返回就可以。
2. 可以，道理其实跟1类似，只是这里是同一个线程而已（相当于自己等自己）

展开

2021-08-18

问题1: 不可以放在栈上, 因为一旦该函数调用返回后, 栈的内存就会被回收。

Bruce
2021-08-18

订阅了课程，如何进群呀

展开

编辑回复: 编辑代回复一下这条：在课程首页，有一个“Rust语言入门交流群”链接，点击扫二维码就能加入啦

2

1031
2021-08-18

对无法确定大小和大小可以改变作为堆和栈的区别，我是存疑的，栈上也可以

作者回复: 可以看我之前的回复。是的，栈上可以存放大小不确定的数据。

kimi
2021-08-17

两个思考题本质上是一个问题，答案都是可以。第二个问题大部分同学的回答都是可以，如果第二个问题可以，是可以推导出第一个问题也是可以的，因为栈上的变量和数据结构本质上都是栈上的一段内存区域，当然在栈上分配的数据结构得是定长的，如果这段内存区域可以使用指针引用，那这个指针就是可以在多个线程之间共享的。

...

展开

作者回复: 第一个问题：如果值在线程 A 结束的时候释放，而此刻线程 B 对其的引用还存在，则会存在 use after free 的危险。

Yuk亮

2021-08-17

貌似问题1大家都有不同意见。我认为是可以访问的，理论上只要有变量地址，那就可以访问。是否安全则是另外一个问题。实际上多线程访问一个栈上的变量的场景也不少，比如在任务并行执行的场景中，业务处理的线程创建一个atmoic的栈变量作为计数器，把需要并行的任务与这个栈变量的地址传递到线程池中，当任务完成，把计数器+1。业务线程则在检测计数器是否达到预期，以达到等待所有任务完成的目的。

展开

作者回复: 嗯，如果不考虑安全，那是都可以访问。

邵俊达

2021-08-17

请问「栈」那节图中的 FP 是什么意思？可以顺便讲一下那张图吗？

作者回复: FP 是 frame pointer，指向栈帧的起始位置。还有一个 SP（stack pointer）指向栈顶，它会随着栈的操作而变化。在开一个新的栈帧的时候，在栈顶会压入 callee 函数的参数，caller 返回地址，以及 caller FP。

啥都不会的二胡

2021-08-17

1. 如果有一个数据结构需要在多个线程中访问，可以把它放在栈上吗？为什么？
从理论上来说可以存放于栈上。函数被执行时系统会在栈中为之分配栈帧，只要函数不结束运行，为其分配的栈帧不会被回收。而线程有都各自具有独立的栈空间，假设我们创建多个线程，每个线程都有一个入口函数，如果这些入口函数都进行死循环，则入口函数不会退出，栈帧也不会被回收。我们在入口函数中定义能够在编译时大小明确的局部变量...

展开

作者回复: 1. 对，然而这种情况需要开发者来确保其安全性。
2. 对。

Geek_97036e

2021-08-17

请教陈老师：

关于可执行文件加载string literal 到内存，如果不同的两个函数，foo() 和bar()中，各自都用到了一个string literal “hello”，那么编译器从从可执行文件.rodata 中加载内存，是加载一份”hello”，还是两份？

展开

作者回复: 是一份。它们指向 .rodata 中同样的地址。你可以用这段代码测试一下：

```rust
fn main() {
    let x = "hello world";
    println!("x = {:p}", x);
    foo();
}

fn foo() {
    let v = "hello world";
    println!("v = {:p}", v);
}
```

1

abeng

2021-08-17

1. 如果有一个数据结构需要在多个线程中访问，可以把它放在栈上吗？为什么？
理论上是可以的，也就是Cross-thread Stack Access，比如通过指向该数据结构的全局指针Ptr0，Thread1中初始化数据结构，地址保存到Ptr0，然后Thread2中访问Ptr0。但是，这么作死何必呢？...

展开

作者回复: 1. 赞「作死」:)
2. 对